Стандарт оформления и документирования исходного кода

Структура стандарта

Содержание данного раздела имеет следующую структуру:

• **Правилом** называется минимальная единица стандарта, частично или полностью описывающая форму представления элемента языка PHP в рамках данного стандарта. *Правило* должно отвечать всем или большинству (в случае аргументированного отступления от конкретного *критерия*) *критериев оценки*, описанным ниже. Каждое *правило* должно дополняться аргументами, приведшими к появлению данного *правила* (со ссылкой на *критерии оценки*) и примерами реализации в виде PHP кода;

• Все *правила*, относящиеся к одному элементу языка PHP (на пример к именам переменных или строковым литералам) называются **формой представления** этого элемента и должны записываться на отдельной странице стандарта;

• *Формы представления* логически связанных элементов языка PHP группируются в **разделы**;

• Стандарт имеет иерархическую структуру с тремя уровнями иерархии:

1. Корневые узлы, имена которых соответствуют языку, элементы которого описываются стандартом;
2. Узлы *разделов*, группирующих логически связынне элементы языка;
3. Страницы стандарта, содержащие *формы представления* конкретных элементов языка, относящихся к данному *разделу*. Имена этих узлов должны соответствовать именам элементов, описываемых в них.

Критерии оценки

Критерии оценки позволяют определить верность используемого *правила*. Если два *правила* описывают одну и ту же часть элемента, предпочтение отдается тому, который отвечает наибольшему числу *критериев*. *Критерии* расположены в порядке убывания приоритета, это значит, что если один критерий вступает в конфликт с другим (что называется *конфликтом критериев*), предпочтение отдается тому, номер которого ниже. В случае, если существует два *правила* описывающих одну и ту же часть элемента, при этом одно правила создает *конфликт критериев*, а другое нет, предпочтение отдается второму правилу.

1. **Читабельность** - *правило* должно подчеркивать логическое назначение элемента. Данный критерий требует писать программный код так, чтобы его было легко читать и понимать в дальнейшем;
2. **Простота** - программист должен выполнить минимальное число операций для записи элемента. Данный критерий позволяет программисту сэкономить время для интеллектуальной работы, связанной с проектированием кода, за счет уменьшения количества времени, отводимого на написание кода. Кроме того, простые *формы представления* элемента проще реализовать в других языках, что снижает время переучивания при переходе между языками (стандартами);
3. **Эффективность** - *правило* не должно снижать эффективность элемента, то есть алгоритм, в котором используется данный элемент или группа элементов, не должен выполнять больше работы при использовании *правила*. Данный критерий призван уберечь программиста от излишнего "жертвования" аппаратными ресурсами.

Можно заметить, что все критерии конфликтуют между собой. Часто для достижения максимальной эффективности, приходится жертвовать читабельностью, а повышение читабельности, снижает простоту и т.д. Для разрешения такого рода конфликтов необходимо учитывать приоритетность критериев, а так же критически относиться к каждому *правилу* данного стандарта.

PHP

Литералы

Строковые литералы

1. Строковый литерал записывается в одинарных кавычках (')

В связи с тем, что в языке PHP возможна запись строковых литералов с использованием двойных (с интерпретацией) и одинарных (без интерпретации) кавычек, предпочтение отдается одинарым, так как это позволяет избежать интерпретации строковых литералов, когда этого явно не нужно (критерий *эффективности*). Так же включение переменных в строковые литералы более очевидно в случае использования конкатенации, чем в случае интерпретации (критерий *читабельности*). Данное правило не отвечает критерию *простоты*, так как требует использования большего числа символов, в случае включения переменных в строковый литерал, но так как других альтернатив в языке PHP нет, данное решение отвечает большему числу критериев.

```
$str = 'String';
$char = 'A';
```

2. Если строковый литерал следует объединить с данными из других источников, его следует явно конкатенировать с этими данными по средствам оператора Точка, а не использовать двойные кавычки

Данное правило прямо следует из правила 1, так как при использовании иного подхода к включению переменных в строковые литералы, это привело бы к противоречию правил.

```
$var = ' ';
$message = 'Hello' . $var . 'world';
```

3. Двойные кавычки (") для обозначения строковых литералов могут применяться только в тех случаях, когда необходимо использовать специальные символы языка PHP (возврат каретки, перенос строки и т.д.). В этом случае специальные символ записывается отдельно и объединяется со строкой с помощью конкатенации

Данное правило следует из ограничений самого языка PHP. Другого способа создания специальных символов в виде литералов в нем просто не существует.

```
$message = 'Hello world'."\n";
```

4. Строковый литерал не переносится между строками файла

В случае применения автоматического переноса строк файла (средствами текстового редактора), приводит к "потере" конкретной строки в файле при чтении программистом (критерий *читабельности*), а в случае ручного выставления переносов, требует от программиста дополнительных действий (критерий *простоты*). Оба подхода никак не сказываются на эффективность алгоритма, потому критерий *эффективности* не берется в расчет. Однако следует отметить, что применение механизма автоматического переноса строк, выходящих за границы окна текстового редактора, позволяется с целью **временного** чтения его содержимого, не требующего прокрутки окна по горизонтали.

```
$badStr = 'Hello
world'; // Пример нарушения правила.
```

5. Не следует использовать строковый литерал в качестве операнда логического выражения, вместо этого его следует присвоить константе класса, к которому этот литерал относится

Данное правило призвано исключить использование "магических строк". *Читабельность* кода, в котором используются не связанные с конкретной сущностью строки, резко снижается. Так же это исключает возможность дальнейшего изменения строкового литерала без изменения некоторой логики кода, зависящего от него. Эффективность кода, не использующего "магические строки" не снижается, но критерием *простоты* приходится принебречь в связи с приоритетом критерия *читабельности*. Наиболее приоритетным местом размещения "магических строк" являются константы классов, к которым они относятся.

```
class Message{
  const INFO = 'info';
  const WARNING = 'warning';
  const ERROR = 'error';

  ...
}
```

Логические литералы

1. Логические литералы (true, false) записываются в нижнем регистре

Так, как регистр логического литерала не имеет значение, данное правило требует от программиста меньше действий при реализации (исключение операции перевода регистра), что отвечает критерию *простоты*. *Читабельность* логических литералов при использовании нижнего регистра, соответствует читабельности при записи литерала с заглавной буквы, но второй вариант конфликтует с критерием *простоты*, за что был исключен. Эффективность скрипта, записанного с любой формой представления логических литералов не меняется, потому критерий *эффективности* не учитывается.

```
$t = true;
$f = false;
```

Литерал NULL

1. Литерал NULL записывается в нижнем регистре

Так, как регистр литерала null не имеет значение, данное правило требует от программиста меньше действий при реализации (исключение операции перевода регистра), что отвечает критерию *простоты*. *Читабельность* литерала null при использовании нижнего регистра, соответствует читабельности при записи литерала с заглавной буквы, но второй вариант конфликтует с критерием *простоты*, за что был исключен. Эффективность скрипта, записанного с любой формой представления литералов null не меняется, потому критерий *эффективности* не учитывается.

```
$n = null;
```

Операторы

Инициализация перемнных и свойств класса

1. Оператор инициализации (=) записывается с использованием обрамляющих пробелов

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
$var = 1;
```

*Арифметические операторы + - * / %*

1. Арифметические операторы (+,-,*,/,%) записываются с использованием обрамляющих пробелов

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
$var = 1 + 1;
```

Операторы инкремента и декремента

1. Операторы инкремента и декремента (++, --) не зависимо от формы (префиксные и постфиксные) записываются без использования разделяющего пробела

Данное правило основывается на предположении, что операторы инкремента и декремента более точно выражают свою урнарную природу (использование одного операнда) при совмещении их с операндом (критерий *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
$var++;
--$var;
```

Логические операторы И ИЛИ НЕ

1. Логический оператор И записывается с использованием слова &&

Данная запись включает меньше символов, чем AND, что отвечает критерию *простоты*. Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if($a && $b){
  ...
}
```

2. Логический оператор ИЛИ записывается с использованием слова ||

Данное правило исходит из правила 1 и призвано использовать единый формат записи (либо AND, OR, либо &&, ||) с целью повышения *читабельности*. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if($a || $b){
  ...
}
```

3. Логический оператор НЕ записывается с использованием слова !

Данная запись включает меньше символов, чем NOT, что отвечает критерию *простоты*. Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if(!$a){
  ...
}
```

4. Логические операторы записывается с использованием обрамляющих пробелов за исключением оператора НЕ, который не отбивается пробелом от выражения

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

Данное правило основывается на предположении, что оператор НЕ более точно выражают свою урнарную природу (использование одного операнда) при совмещении их с операндом (критерий *читабельности*). Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if($a && $b || !$c){
}
```

```
if($a != $b){
  ...
}
```

2. Операторы сравнения записывается с использованием обрамляющих пробелов

Данное правило позволяет визуально разделить операнды, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск операндов в выражении. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
if($a >= $b || $b != $c){
}
```

Логические выражения

1. В случае, если логическое выражение включает более двух подвыражений, следует оценить его читабельность и, возможно, выделить подвыражения в переменные с "говорящими" названиями

Данное правило призвано повысить *читабельность* сложных логических выражений. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
// Пример нарушения правила. Слишком низкая читабельность выражения.
if(elementIndex < 0 || elementIndex > MAX_ELEMENTS || elementIndex == lastElementIndex){
  ...
}
```

```
// Пример следования правилу.
$isOutOfBounds = elementIndex < 0 || elementIndex > MAX_ELEMENTS;
$finished = elementIndex == lastElementIndex;
if($isOutOfBounds || $finished){
  ...
}
```

Тернарный логический оператор ?:

1. Тернарный логический оператор записывается без использования разделяющего пробела слева для символа ?, но с использованием разделяющего пробела с права от этого символа

Данное правило призвано повысить читабельность оператора за счет отделения условного выражения от тела. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

2. Тернарный логический оператор записывается с использованием обрамляющих пробелов для с имвола :

Данное правило позволяет визуально разделить ветви оператора, сделав их более заметными (критерий *читабельности*). Так же правило упрощает поиск значений ветвей в операторе. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

3. Условие тернарного логического оператора записывается с круглых скобках

Данное правило позволяет визуально выделить логическое выражение оператора (критерий читабельности) и избежать ошибок, связанных с приоритетом выполнения. Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

```
($a == $b)? true : false;
```

Управляющие структуры

Ветвление if

1. Управляющая структура if записывается без отделения от открывающей круглой скобки логического выражения

Данное правило упрощает процесс набора структуры (критерий *простоты*). Перенос выражения на новую строку не выполняется с целью сохранения логической структуры выражения: if, then, else. Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

2. Логическое выражение записывается без использования обрамляющих

пробелов

Отбиение логического выражения от круглых скобок условия нисколько не повышает читабельность кода, а лишь добавляет сложность.

3. Код, относящийся к ветвям, всегда группируется в блоки и отбивается одим стандартным отступом относительно структуры

Выделение кода в блоки и отбиение призвано повысить скорость поиска ветви структуры (критерий *читабельности*). Эффективноть таких выражений не страдает, но от критерия простоты приходится отказаться в силу приоритетности критерия *читабельности*.

4. Открывающая фигурная скобка, начинающая блок кода, ставится сразу после закрывающей круглой скобки или слова else. За ней сразу должен следовать символ новой строки

Данное правило следует из логической структуры ветвления. При переносе открывающей фигурной скобки на новую строку повышается сложность кода (критерий *простоты*), но не увеличивается читабельность. Правило так же приследует идею, что открывающим элементом блока кода является конструкция if(...) и else, которые проще найти в коде, нежели одиночные символы открывающих фигурных скобок (критерий *читабельности*). Критерий *эффективности* в расчет не берется в связи с его неизменностью в альтернативных формах записи.

5. Закрывающая фигурная скобка, заканчивающая блок кода, ставится с новой строки на уровне управляющей структуры сразу после окончания блока кода

Данное правило позволяет четко выделить окончание блока кода структуры (критерий *читабельности*). В связи с тем, что альтернативная версия правила, при котором закрывающая фигурная скобка записывается рядом с else, мало отличается от данного варианта, принятое правило является спорным и может быть изменено в будущем.

6. В качестве альтернативного условия используется слово elseif

Данный вариант записи проще else if (критерий простоты). Критерии *читабельности* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

7. Слово else ставится с новой строки

Аналогично правилу 5.

8. Слово elseif ставится с новой строки

Аналогично правилу 5.

```
if($a > $b){
  ...
}
elseif($a < $b){
  ...
}
else{
  ...
}
```

9. Нормальный путь алгоритма следует располагать в первом блоке структуры, а не в блоке else

Данное правило позволяет программисту быстро выделить нормальный ход алгоритма (начало условия) и обработчики ошибок (конец условия), что повышает читабельность. Критерии *простоты* и *эффективности* в расчет не беруться в связи с их неизменностью в альтернативных формах записи.

```
if(...){
  if(...){
    ... // Нормальный ход.
  }
  else{
    ... // Альтернативный ход.
  }
}
else{
  ... // Альтернативный ход.
}
```